university of groningen / faculty of science and engineering

Extending the QV family of deep reinforcement learning algorithms: DQV2 and DQV-Max2

Bachelor's Project Thesis

Mantas Majeris, s4010817, m.majeris@student.rug.nl,
Supervisor: Dr M. Sabatelli

Abstract: This paper attempts to extend the QV family of Deep Reinforcement learning algorithms, where both the state value function, and the state-action value function are approximated using neural networks. We introduce two new algorithms, DQV2 and DQV-Max2, based on the classical RL algorithms QV2 and QV-Max2. We run multiple experiments in the Cart-Pole, Acrobot, and Mountain-Car environments, where we compare the reward obtained over time by the two algorithms DQV2 and DQV-Max2, as well as several established QV family algorithms, specifically DQV, DQV-Max, and an algorithm which only learns the state-action value function DQN. Preliminary results suggest that DQV-Max2 is comparable to DQV-Max in performance, while DQV2 vastly underperforms with most of the hyperparameter combinations used in this study. However, for some hyperparameter combinations, DQV2 and DQV-Max2 achieve comparable performance to DQV and DQV-Max with greater consistency.

1 Introduction

Reinforcement Learning (RL) is a method of training an agent to act in a particular environment. This is often done by estimating a value function V, which estimates the value of being in a particular state, or an action quality function Q, which estimates the value of performing a certain action in a certain state (Sutton and Barto, 2018). These functions are then used to determine an optimal policy which determines which action an agent should take in any given state. There exist many classical RL algorithms with different update rules for the Q and V functions, such as Q-learning (Watkins and Dayan, 1992) and SARSA (Rummery and Niranjan, 1994). Another type of RL algorithm is the QV family, which attempts to learn both the Q and V functions (Wiering and Van Hasselt, 2009). All of these algorithms are tabular RL algorithms, meaning the functions get updated only for one particular state and one particular action (in the case of the Q function) at a time. However, in complex environments with large (possibly continuous) state spaces, classical RL algorithms fail, as most states are simply not visited enough times. Additionally, in large state spaces, keeping track of all state-action combinations becomes computationally impossible.

In deep Reinforcement Learning (DRL), we attempt to approximate the Q and/or V functions using neural networks (NNs). In the case of the QV family, we approximate both the V function, and the Q function. The DRL extensions of the QV and QV-Max algorithms, DQV and DQV-Max, are described by Sabatelli, Louppe, Geurts, and Wiering (2020), who find that DQV and DQV-Max converge more quickly, and overall outperform the well-known algorithms DQN and DDQN. In this paper, we attempt to use function approximators to extend the QV2 and QV-Max2 classical RL algorithms (Wiering and Van Hasselt, 2009), which are not covered by Sabatelli et al. (2020). The specific update rules and loss functions for each algorithm are described in the next section.

2 Preliminaries

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is defined as a tuple $\langle S, \mathcal{A}, T, R \rangle$, where S is a set of states, $\mathcal{A}$ is a set of actions, $T : S \times \mathcal{A} \times S \to [0, 1]$ is a func-

tion representing the transition probabilities from one state, given an action, to another state. Finally, the reward function $R : S \times \mathcal{A} \times S \to \mathbb{R}$ is a function representing the reward an agent gets for making a particular transition. Additionally, a time counter t is used to keep track of which time-step the MDP is in. An agent is needed to act upon the environment described by an MDP. At each time-step, when the agent is in state $s_t \in S$, it must make some action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of actions available to the agent at state s_t. After making the action, the agent transitions to a successor state s_{t+1}.

In Reinforcement Learning, we are interested in learning an optimal policy $\pi^* : S \to \mathcal{A}$, which would maximize the reward obtained by the agent. To that end, we are often interested in learning a function $Q : S \times \mathcal{A} \to \mathbb{R}$, which gives us the value of taking a particular action in a certain state.

Formally, the Q function under policy π is defined in equation 2.1.

$$Q^\pi(s,a) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \Big| s_t = s, a_t = a, \pi\Big] \quad (2.1)$$

Where $\gamma \in (0,1)$ is a discount factor, meant to account for future rewards, and R_{t+k} is the reward obtained at time-step $t+k$. Essentially, the Q function is the expected total discounted reward if an agent starts at state s, performs action a, and follows policy π afterwards. Another important notion for this project is the V function, defined in equation 2.2.

$$V^\pi(s) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \Big| s_t = s, \pi\Big] \quad (2.2)$$

Where γ is again the discount factor, and the function is defined for a particular policy. The V function is the expected total discounted reward the agent will get by starting at state s, and then following policy π. An optimal policy π^* is one where the Q and V values are maximal, as the Q and V functions are known to satisfy the Bellman optimality equation, shown in equation 2.3 for the V function.

$$V^*(s_t) = \max_a \sum_{s_{t+1}} T(s_t, a, s_{t+1}) \Big[r(s_t, a, s_{t+1}) + \gamma V^*(s_{t+1}) \Big] \quad (2.3)$$

And equation 2.4 for the Q function.

$$Q^*(s_t, a_t) = \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \Big[r(s_t, a_t, s_{t+1}) + \gamma \max_a Q^*(s_{t+1}, a) \Big] \quad (2.4)$$

Meaning we have a formal definition of an optimal policy in terms of the Q and V functions.

In model-free reinforcement learning, iterative update rules are typically used to learn these functions, depending on the precise learning algorithm. These algorithms can be applied with both Monte-Carlo methods, where the function estimations are updated at the end of a learning episode, or with temporal difference (TD) learning, where the functions are updated at each time step. This paper will only make use of TD learning.

2.2 Neural Networks

For a successful DRL application, a suitable function approximator is necessary. One such candidate is an artificial neural network. In fact, Multi-layer perceptrons (MLPs), which is a neural network consisting of linear layers and non-linear activation functions in between the layers, are proven to be universal function approximators (Hornik, Stinchcombe, and White, 1989). As such, an MLP should serve as a suitable function approximator for our V and Q functions, provided the architecture used is both capable of apprixmating the Q and V functions, and does not have too many trainable parameters to train in a reasonable amount of time.

Another type of neural network which can be better used on more complex data is a convolutional neural network (CNN) (LeCun, Bengio, and Hinton, 2015). CNNs serve as feature extractors, and contain much fewer trainable parameters when the network is large. Like MLPs, CNNs are also universal function approximators, though their use is rather redundant if the locality of the input data

has no meaning. As such, CNNs are very powerful tools for approximating the Q and V functions when the input data is visual, but MLPs are preferred on simpler data.

3 Related Work

A number of algorithms have been developed to learn this function, one of the most well known being Q-learning (Watkins and Dayan, 1992). The Q-learning algorithm starts with some initial estimates for the Q values for each state-action pair, and updates them with the rule shown in equation 3.1

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha(r_t + \gamma \max_{q \in \mathcal{A}} Q(s_t, a) - Q(s_t, a_t)) \quad (3.1)$$

Where $\alpha \in (0, 1)$ is the learning rate.

It was shown by Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, et al. (2015) that it is possible to extend classical Q-learning to DRL, and obtain the Deep Q-Network (DQN) algorithm, where the $Q(s, a)$ function in approximated by a function $Q(s, a; \theta)$, where θ is a set of parameters which define a neural network (or some other function approximator). We learn the Q function by tuning θ in an attempt to minimize the loss function described in equation 3.2

$$L(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \Big[(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta))^2 \Big] \quad (3.2)$$

Where D is an experience replay buffer containing trajectories $\langle s_t, a_t, r_t, s_{t+1} \rangle$. The loss function makes use of a technique known as memory replay (Lin, 1992), where observed trajectories are stored in a replay buffer, and possibly re-used for future learning. These trajectories can be sampled in any manner, but in the context of the Arcade Learning Environment (ALE), a uniform sampler U is used (Bellemare, Naddaf, Veness, and Bowling, 2013). The purpose of this loss function is to fit $Q(s_t, a_t; \theta)$ to the TD target y_t (see equation 3.3).

$$y_t = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta) \quad (3.3)$$

However, Q-learning, and its DRL counterpart both suffer from an overestimation bias . The max operator in the update rule/loss function results in overestimated TD targets when noise is present. There have been a number of algorithms developed in an attempt to solve this problem. One of which is Double Q-learning (Hasselt, 2010), which learns two separate Q functions, both of which are mutually dependent on each other for learning. Additionally, Double Q-learning has been proven to function in a DRL context as well (Van Hasselt, Guez, and Silver, 2016), where the two Q functions are approximated with NNs.

In fact, a great deal of attempted improvements to the DQN algorithm have been attempted, with varying degrees of success. Another such attempt is prioritized experience replay (Schaul, Quan, Antonoglou, and Silver, 2015), which attempts to sample more useful transitions more often, as opposed to sampling transitions uniformly. There is also distributional RL (Bellemare, Dabney, and Munos, 2017), which attempts to model the value distribution itself, rather than just the expectation. An agent which integrates these techniques in addition to several others has been proposed by Hessel, Modayil, Van Hasselt, Schaul, Ostrovski, Dabney, Horgan, Piot, Azar, and Silver (2018) in the form of the *Rainbow* agent.

3.1 DQV Family

Of interest to this paper is another attempted improvement over DQN. Specifically, the QV family of algorithms, which in addition to the Q function, attempt to learn the $V : S \to \mathbb{R}$ function, which represents the value of a particular state, and is used to assist in learning the Q function. Several of these QV algorithms are examined by Wiering and Van Hasselt (2009). Specifically, of the QV family, they examine the classical RL algorithms QV, QV2, QV-Max, and QV-Max2, and test them on maze problems, as well as the cart-pole problem.

DRL extensions of the QV and QV-Max are introduced by Sabatelli et al. (2020), where the V function is also approximated with a neural network, defined by a set of parameters Φ. The DRL extension of QV-learning (DQV) uses the loss function shown in equation 3.4 to learn the V function.

$$L(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Big[(r_t + \gamma V(s_{t+1}; \Phi^-) - V(s_t; \Phi))^2\Big] \quad (3.4)$$

Where Φ^- is a copy of the Φ network, but we lock in the parameters, and consider it to be a constant when differentiating with respect to Φ.

The Q funciton is learned with the loss function shown in equation 3.5.

$$L(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Big[(r_t + \gamma V(s_{t+1}; \Phi^-) - Q(s_t, a_t; \theta))^2\Big] \quad (3.5)$$

DQV-Max, the DRL extension to QV-Max, learns the V function with the loss function shown in equation 3.6, while the Q function is learned using equation 3.5, same as DQV.

$$L(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Big[(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta) - V(s_t; \Phi))^2\Big] \quad (3.6)$$

These loss functions resemble the update rules for the classical DQV and DQV-Max algorithms. Both classical algorithms use the same update rule for the Q function, shown in equation 3.7.

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha(r_t + \gamma V(s_{t+1}) - Q(s_t, a_t)) \quad (3.7)$$

Where $\alpha \in (0, 1)$ is the learning rate.

The update rules for the V functions are showing in equations 3.8 and 3.9 for DQV and DQV-Max respectively.

$$V(s_t) := V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - V(s_t)) \quad (3.8)$$

$$V(s_t) := V(s_t) + \alpha(r_t + \gamma \max_{a \in A} Q(s_t, a) - V(s_t)) \quad (3.9)$$

3.2 New DRL algorithms

Wiering and Van Hasselt (2009) describe the QV2 and QV-Max2 algorithms. QV-Max2 and QV2 both have the same update rule as QV and QV-Max for the Q function (see equation 3.7), but different V update rules. The V update rule for QV2 is described in equation 3.10.

$$V(s_t) := V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - Q(s_t, a_t)) \quad (3.10)$$

Where α is the learning rate. The QV-Max2 V update rule is shown in equation 3.11.

$$V(s_t) := V(s_t) + \alpha(r_t + \gamma \max_{a \in A} Q(s_t, a) - Q(s_t, a_t)) \quad (3.11)$$

4 Methods

4.1 Loss functions used

The update rules for QV2 and QV-Max2 do not easily convert to loss functions. Directly converting the classical update rules to loss functions would result in a constant loss function for the V network, as the term being fit to the TD-target would not depend on Φ. To solve this problem, we propose the DRL algorithms DQV2, and DQV-Max2, which use the same Q loss function as DQV and DQV-Max, and use the V loss functions in equations 4.1 and 4.2 respectively.

$$L(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Big[(r_t + \gamma V(s_{t+1}; \theta) - w \cdot Q(s_t, a_t; \theta) - (1 - w) \cdot V(s_t; \Phi))^2\Big] \quad (4.1)$$

$$L(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Big[(r_t + \gamma \max_{a \in A} Q(s_{t+1}, a; \theta) - w \cdot Q(s_t, a_t; \theta) - (1 - w) \cdot V(s_t; \Phi))^2\Big] \quad (4.2)$$

Where $w \in (0, 1)$ is what we call the Q-weight. The Q-weight will determine how influential the Q function is when tuning the V function estimate. A larger w will result in the V function estimate being more static, as the loss will depend less on the V function. Meanwhile, as w becomes closer to 0, the loss functions become closer and closer to those of DQV and DQV-Max. The reason this setup is expected to work is because when the Q and V functions are learned perfectly, $V(s) = \max_a Q(s, a)\ \forall s \in S$. Therefore, when both

the V and Q terms are added to the loss function, as long as their coefficients add up to 1, the loss function should lead to a useful approximation of V.

4.2 NN architecture

We test the algorithms using a Multi-layer perceptron (MLP) as the function approximator for the Q and V functions. we used the same MLP architecture as Kurban (2020). The QV architectures used two such DQN architectures with different output dimensions: one for the Q function, the other for the V function.

Additionally, a CNN architecture was also implemented for the option to use all of the algorithms on the Atari games. The architecture is the same as the one used by Mnih et al. (2015). Due to time constraints, in addition to the fact that learning in environments as complex as the Atari games takes much longer, sufficient experiments were not run using the CNN architecture. However, the implementation will be included with the code.

The full code can be found in `https://github.com/MantasMajer/DQV2-and-DQV-Max2`

4.3 experimental setup

For each of the three environments, We run 10 training runs per algorithm. One training run consisted of 500 episodes. The action is selected using the ϵ-greedy method, where the agent has a probability of ϵ to pick a random action, and behave greedily otherwise. A decaying value of ϵ was used, with 0.99 as the decay rate, and the minimum ϵ value being 0.01. The hyperparameters are given in table 4.1.

Learning rate α	0.001
ϵ	0.5
Discount factor γ	0.99
replay buffer size	65536
replay batch size	16
Q-weight w	0.5

Table 4.1: Hyperparameters used for the experiments

We keep track of two measurements. Firstly, the reward obtained during each training episode. Secondly, because we are not necessarily interested in the most trained algorithm, just the best performing, we attempt to keep track of the best version of the model we had by periodically running a pure greedy strategy on what we expect to be the best version of the model so far. The way we estimate the best model is by looking at the training performance, and seeing if it is better than the performance of the best model. If it outperforms the best model, we run a pure exploitation run on the newest model, and if the average of the two runs is greater than the best model, we replace the best model with the current one. Every 10 episodes, we run the best model with a pure exploitation strategy 3 times, keep track of the average total reward from the three runs, and remember this average as the best model's performance. Note, that we do not train the model when we do pure exploitation runs.

4.4 Environments

We used three environments from the OpenAI gym (Brockman, Cheung, Pettersson, Schneider, Schulman, Tang, and Zaremba, 2016). Specifically, the Cart-Pole, Acrobot, and Mountain-Car environments. All of these are basic control tasks with relatively small action and state spaces, so they should not be particularly difficult to learn. As such, they should serve as a good initial testing benchmark for the new algorithms before they are tested on more complex vision-based environments.

4.4.1 Cart-Pole

This environment simulates a cart which is able to move left or right, and must balance a pole such that it does not fall down (see figure 4.1). Cart-Pole has an action space of size 2, where the available actions are to take a step to the left, or the right. States are defined by four continuous values:

1. **Cart position**, range: $(-4.8, 4.8)$
2. **Cart velocity**, range: $(-\infty, \infty)$
3. **Pole angle**, range: $(-0.418, 0.418)$
   0.418 radians is approximately 24 degrees.
4. **Pole angular velocity**, range: $(-\infty, \infty)$

An episode will terminate if the cart leaves the $(-2.4, 2.4)$ range, or if the pole's angle leaves the

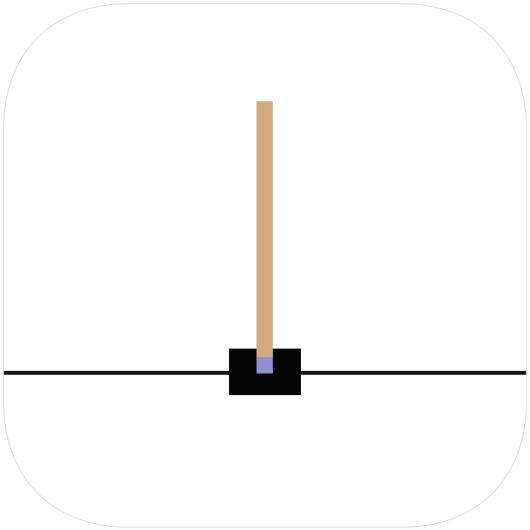

Figure 4.1: An image illustrating the Cart-Pole environment. `https://github.com/ganeshjha/Cartpole`

$(-0.2095, 0.2095)$ range. The agent will obtain a reward of 1 for each time-step the episode does not terminate. Rewards are capped at 500.

4.4.2 Acrobot

This environment has a slightly larger action and state space. The problem consists of two links with an actuated joint between them, and with one of the links having its other joint fixed in place. The agent should apply torque to the actuated joint to swing the other end up above a certain threshold (see figure 4.2).

The agent has three available actions: apply -1, 0, or 1 torque to the actuated joint. The state is defined by six values, all relative to two angles θ_1 and θ_2, where θ_1 is the angle between the first link and a line pointing downward, and θ_2 is the angle between the two links. The state is then defined by the sines, cosines, and angular velocities of θ_1 and θ_2.

The agent will incur a reward of -1 for each time-step it does not reach the goal, and an episode will terminate when the goal is reached. The lower bound on the reward for an episode is -500.

Figure 4.2: An image illustrating the Acrobot environment. `https://talukder88.github.io/acrobot.pdf`

4.4.3 Mountain-Car

This environment consists of a mountainous track, and a car which must go up a hill (see figure 4.3)

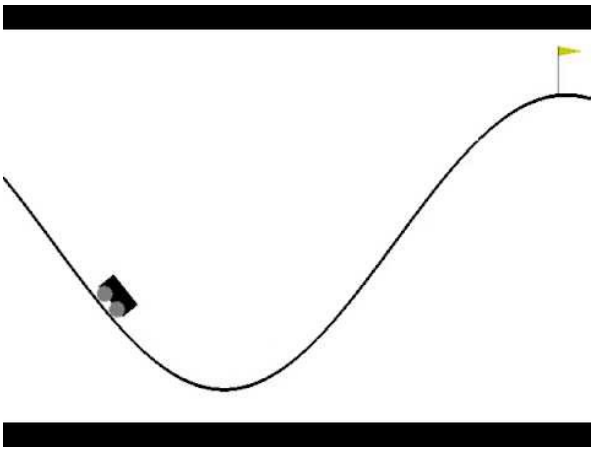

Figure 4.3: An image of the Mountain-Car environment. `https://github.com/kumarnikhil936/q_learning_mountain_car_openai`

The car must reach the top of the hill, but does not have enough power to do so, so it must go back and forth to get an increasingly large initial velocity when it starts to drive up the hill.

There are three actions available to the agent: accelerate to the left, the right, or do not accelerate. The state is defined by the car's position on the x axis, and its velocity.

Like with Acrobot, the agent incurs a reward of -1 for each time-step it does not reach the goal, and reaching the goal terminates an episode. Episodes have a lower bound of -200.

4.5 Additional investigations

Another aspect of the new algorithms that may be worth investigating is the effect of the Q-weight on the performance of the algorithms. As such, we ran two additional experiments where we tested DQV against DQV2 with five different values for the Q-weight. We ran an analogous experiment on DQV-Max2: DQV-Max against five iterations of DQV-Max2. For the specific values of the Q-weight, we used the values 0.1, 0.2, 0.3, 0.4 and 0.5. Using a value of the Q-weight which is close to 1 would result in a loss function which is not very sensitive to the network parameters being tuned. These experiments were only run on the Cart-Pole environment.

5 Empirical Results

5.1 Main experiment

The experimental results are given in several figures, which depict the reward obtained during training, and the reward obtained at 10 episode increments under a pure exploitation strategy. Additionally, numerical results were also calculated. To formally compare the performance of the algorithms, the areas under the curves were calculated to obtain the total reward obtained throughout training. The total rewards can then be averaged across all experiments, and standard deviations can be computed.

The mean total rewards μ, the standard deviations σ on the Cart-Pole environment are shown in table 5.1. Note, that the values in the tables are rounded down to the nearest integer. This is to make the results more readable, and because the values are quite large, and are unlikely to be affected by small fractional deviations. Additionally, the exploitation runs were made every 10 episodes, meaning the scale of the total rewards obtained will be significantly smaller than the training runs.

algorithm	μ	σ	μ'	σ'
DQN	89065	22212	20202	1150
DQV	193787	11248	21681	862
DQV-Max	187632	7455	21995	845
DQV2	18183	19616	19339	3741
DQV-Max2	155358	22549	21756	1077

Table 5.1: Table showing the mean total rewards and standard deviations across experiments for each algorithm on the Cart-Pole problem. The values are rounded down to integers. μ and σ refer to the training results, while μ' and σ' refer to the exploitation results. The best result in a given measure is highlighted in green, while the second best is highlighted in yellow.

We can see that DQV and DQV-Max perform the best overall, but DQV-Max2 performs similarly in the pure exploitation measure. DQV2 appears to significantly underperform in both measures.

The results for the Acrobot experiment are shown in table 5.2.

algorithm	μ	σ	μ'	σ'
DQN	-65387	1009	-16074	674
DQV	-61697	2641	-5277	464
DQV-Max	-62001	1427	-5351	381
DQV2	-221536	36498	-15277	6871
DQV-Max2	-62611	1365	-5710	369

Table 5.2: Table showing the results on the Acrobot experiment in the same manner as table 5.1 does for the Cart-Pole environment.

During training, all the algorithms appear to perform similarly, with the exception of DQV2, which significantly underperforms, with a much lower average total reward, and a very high standard deviation. Looking at the pure exploitation runs, we see that DQN's performance drops noticeably, while DQV, DQV-Max, and DQV-Max2 stay roughly the same.

The results on the Mountain-Car environment are given in table 5.3.

algorithm	μ	σ	μ'	σ'
DQN	-85936	2275	-8993	224
DQV	-86134	2266	-8270	426
DQV-Max	-86429	2418	-8332	421
DQV2	-99829	130	-10000	0
DQV-Max2	-85532	2742	-8127	488

Table 5.3: Table showing the results on the Mountain-Car experiment in the same manner as table 5.1 does for the Cart-Pole environment, and table 5.2 does for Acrobot.

We see a similar pattern on the Mountain-Car environment as on the Acrobot environment, where DQV2 underperforms massively, even apparently not managing to learn the problem whatsoever. DQN performs similarly to the algorithms other than DQV2 on training, but much worse on the exploitation measure, while the remaining three algorithms perform similarly well.

5.2 Q-weight experiments

Results regarding the effect of the Q-weight on the performance of DQV2 are reported in figure 5.4 and table 5.4 in the same manner as for the main experiment. The same results for DQV-Max2 are reported in figure 5.5 and 5.5.

We see that the Q-weight affects the two algorithms quite differently. On DQV2, it appears that a larger Q-weight largely results in poorer performance on the Cart-Pole problem. Although it appears that for small Q-weights such as 0.1, performance is similar to DQV, and the results in this case even learn the problem slightly faster, and with a smaller standard deviation among total rewards across experiments.

DQV-Max2 appears to be affected a lot less than DQV2 by the value of the Q-weight up to $w = 0.5$ on the Cart-Pole problem. The total reward during training appears to decrease somewhat with increasing Q-weights, but the exploitation measure does not appear to change much, and all the versions seem to be able to learn the problem fully. Like with DQV2 compared to DQV, it appears that for small Q-weights, we see DQV-Max2 gets a slightly smaller standard deviation compared to DQV-Max.

Q-weight	μ	σ	μ'	σ'
0 (DQV)	181111	24453	20605	2400
0.1	195816	7527	22135	654
0.2	151703	54123	21330	1984
0.3	107572	66997	20462	3039
0.4	25263	21170	18437	4085
0.5	13672	6316	12676	6939

Table 5.4: Table showing the mean total rewards and standard deviations across experiments of the DQV2 algorithm for different values of Q-weight on the Cart-Pole problem.

Q-weight	μ	σ	μ'	σ'
0 (DQV-Max)	184656	14956	21851	1675
0.1	188584	7428	22220	597
0.2	181828	6583	21697	621
0.3	176076	16547	22533	459
0.4	167083	13981	21648	900
0.5	149189	18105	21214	1208

Table 5.5: Table showing the mean total rewards and standard deviations across experiments of the DQV-Max2 algorithm for different values of Q-weight on the Cart-Pole problem.

6 Discussion

The results, if replicable, have some interesting implications for the QV family of DRL algorithms. Firstly, the results from these experiments suggest that when the Q-weight is small, the total rewards obtained will be more consistent (i.e. smaller standard deviation) without a noticeable decrease in performance. This appears to be the case for both DQV2 and DQV-Max2. On the other hand, large Q-weights appear to hurt the performance, more so on DQV2 than DQV-Max2.

It is difficult to say why precisely the Q-weight has this effect on the learning, but we can examine the factors that play into the algorithms' learning. The crucial difference between DQV2 and DQV, and between DQV-Max2 and DQv-Max is the fact that the loss function for the V network is less sensitive to the V network Φ itself, and that $Q(s_t, a_t; \theta)$ plays into learning $V(s_t; \Phi)$. As such, we would expect performance to drop as the Q-weight gets close to 1, since the loss function becomes increasingly independent of θ, with it being completely constant

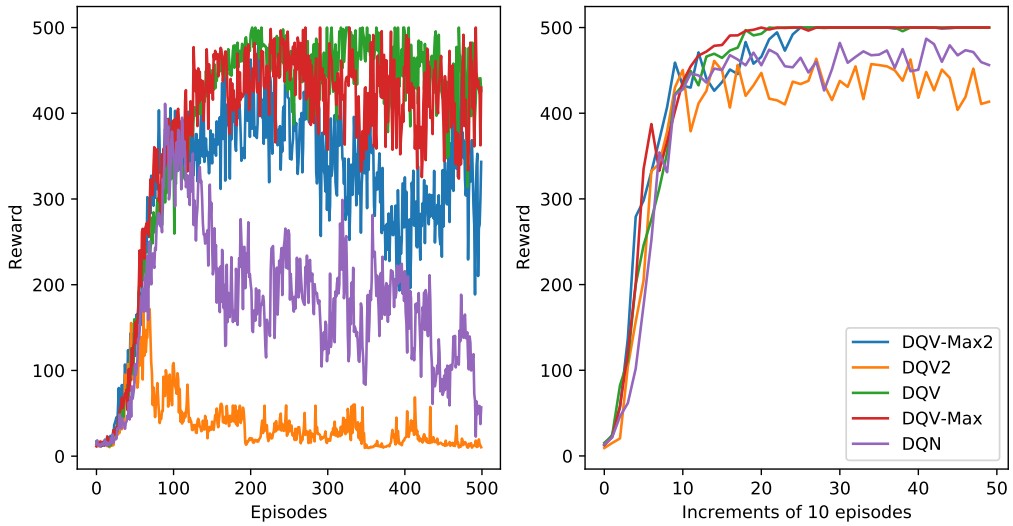

Figure 5.1: Learning curves for the cart-pole problem. The left graph shows the agent's obtained reward for each episode while training, while the graph on the right shows the performance of the suspected best model using a pure exploitation strategy every 10 episodes.

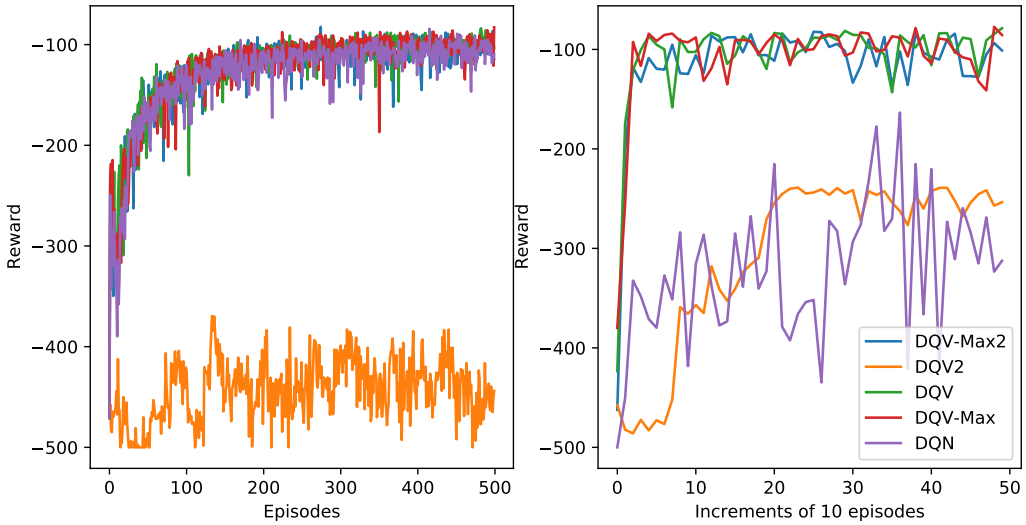

Figure 5.2: Learning curves for the acrobot problem. The left graph shows the agent's obtained reward for each episode while training, while the graph on the right shows the performance of the suspected best model using a pure exploitation strategy every 10 episodes.

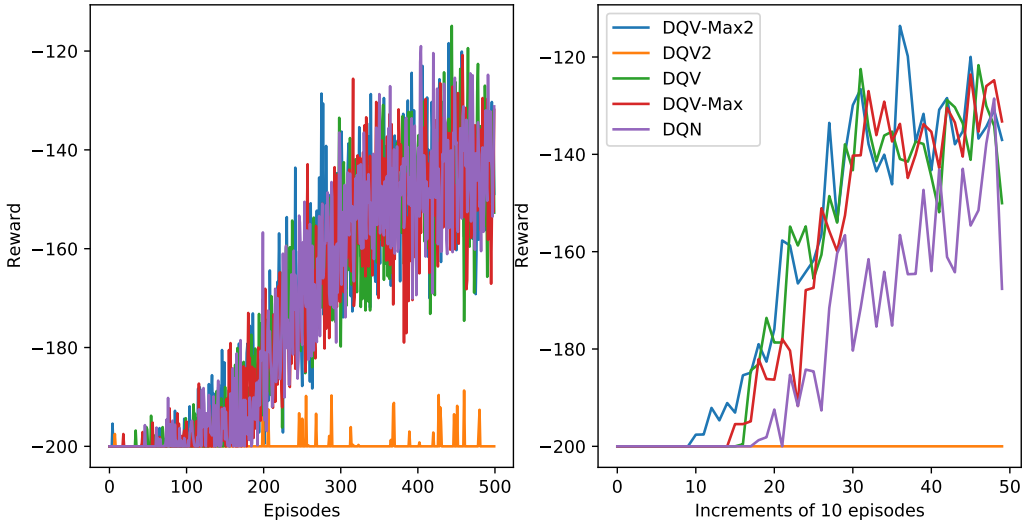

Figure 5.3: Learning curves for the Mountain-Car problem. The left graph shows the agent's obtained reward for each episode while training, while the graph on the right shows the performance of the suspected best model using a pure exploitation strategy every 10 episodes.

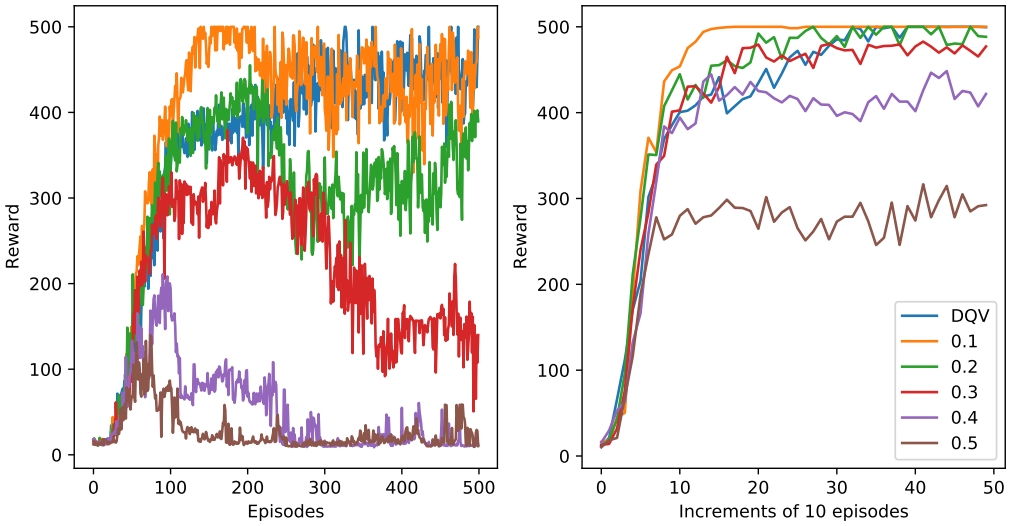

Figure 5.4: Learning curves of DQV2 with varying Q-weights on the Cart-Pole problem. The numerical values in the legend refers to the Q-weight of that iteration of the DQV2 algorithm

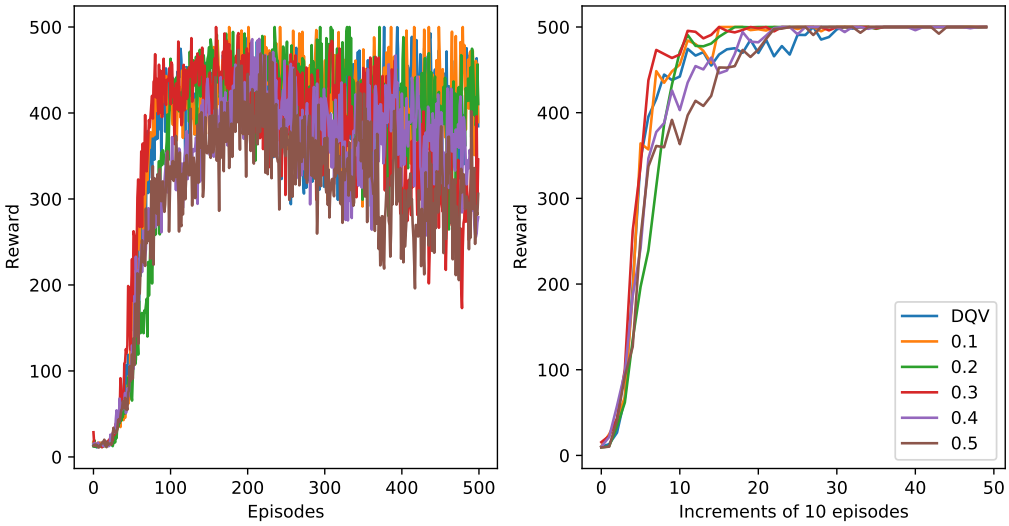

Figure 5.5: Learning curves of DQV-Max2 with varying Q-weights on the Cart-Pole problem. The numerical values in the legend refers to the Q-weight of that iteration of the DQV-Max2 algorithm

when $w = 1$. While the extreme case where w is close to 1 is an obvious case, the results suggest that performance starts dropping sooner, especially for DQV2.

As for why DQV2 seems to suffer from an increased Q-weight more than DQV-Max2, it is hard to say for sure. Additional studies into these algorithms would need to be conducted, including practical experiments, and possibly a thorough mathematical analysis of the subject. It is worth noting, that the tabular algorithm QV2 (Wiering and Van Hasselt, 2009) appeared to perform just as well as QV. We would expect most potential explanations for why DQV2 underperforms to apply to QV2 as well. It is of course possible that the problem is only present with a function approximator present, or that the introduction of the Q-weight makes DQV2 not fully faithful to QV2. These claims would need to be formally investigated by another study.

6.1 Limitations of the research

This research has some significant limitations and should not be taken at face value. For one, all the results are based off of 10 experiments per algorithm. This makes finding reliable statistical significance difficult. To confirm or deny the suggestion that a small non-zero Q-weight leads to more consistent learning would require replication of these results over more training runs.

Additionally, the experiments do not examine every aspect of the DQV2 and DQV-Max2 algorithms. For one, the Q-weights are only varied on the Cart-Pole environment. To make general statements about the effect of the Q-weight on learning in general, experiments like the ones performed in this study should be conducted on other environments as well. The Acrobot and MountainCar environments would be a logical next step, but it would be important to conduct experiments on more complex environments such as Atari games, or other vision-based tasks with a high-dimensional state spaces.

It would also be useful to examine more granular increments of w. This study examines increments of 0.1 up to 0.5, and finds results that suggest it is not useful to use a Q-weight close to 0.5. As such, a study may want to use different increments, such as 0.05 up to 0.25. It may also be a worthwhile idea to investigate what would happen with a dynamic Q-weight which changes over time.

7 Conclusions

In this study DRL extensions to two existing RL algorithms were introduced. Due to the small number of experiments, strong conclusions should not be drawn from this study. That said, this preliminary research suggests that introducing a Q-weight like the one described in this paper could potentially increase consistency in performance. Of course, larger scale experiments, as well as experiments on more complex environments should be run before this influence of the Q-weight can be confirmed or denied. Overall, the DQV-Max2 algorithm appears to be a functional DRL algorithm, while DQV2 likely needs a specific Q-weight. However, if the performance of DQV2 is truly hurt by the introduction of the Q-weight, it simply becomes a less effective version of DQV.

References

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279, 2013.

Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR, 2017.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Rita Kurban. Deep q learning for the cartpole, Apr 2020. URL `https://towardsdatascience.com/deep-q-learning-for-the-cartpole-44d761085c2f`.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

A Rummery, G and M Niranjan. On-line q-learning using connectionist systems. 37:14, 1994.

Matthia Sabatelli, Gilles Louppe, Pierre Geurts, and Marco A Wiering. The deep quality-value family of deep reinforcement learning algorithms. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Marco A Wiering and Hado Van Hasselt. The qv family compared to other reinforcement learning algorithms. In *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 101–108. IEEE, 2009.